Testing Framework and Performance Results

Dr Christopher Jones
CCE IOS

4 November 2020

Testing Framework Purpose

- Mimic the characteristics of a HEP data processing framework
  - Similar multi-threaded behavior
  - Similar I/O behavior
  - Should reasonably behave like CMS, ATLAS and DUNE frameworks

- Easily try different I/O implementations
  - Choose what to use via command line arguments

- Experiment agnostic
  - With ability to read actual experiment ROOT files
    - ROOT will dynamically load serialization/deserialization plugins as needed

- Make it easier to perform performance measurements
  - I/O performance
  - threaded scaling performance

Review of ROOT

- ROOT is a C++ toolkit used in HEP

- Relevant parts of ROOT for CCE-IOS
  - A serialization mechanism for C++ objects
  - A file format to store the serialized objects

- ROOT output and concurrency
  - ROOT's file API allows only 1 thread to interact with a file at a time
  - Internally ROOT uses TBB tasks
    - Can compress different *Data Products* concurrently (serialization is still sequential)

HEP Framework Mimicry

- Only deals with processing of *Events*
- An **Event** is just a collection of *Data Products*

- **Data Product**
  - Can be any C++ type
  - Each Data Product can be accessed independent of all other Data Products

- **Source**
  - Mimics reading of Events

- **Outputer**
  - Mimics writing of Events

- **Waiter**
  - Mimics processing of Events

Processing Events

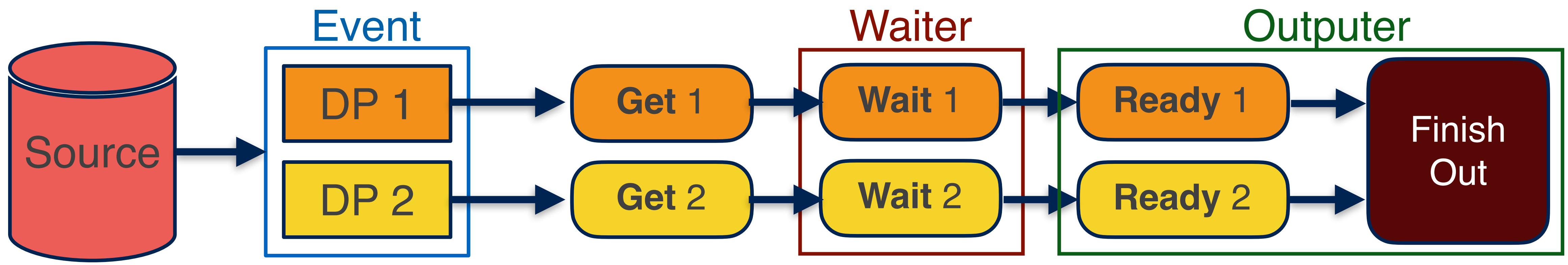
- Specify maximum number of threads to use
  - Use Intel's Threading Building Blocks to control threads
    - Used by CMS, ATLAS, DUNE and ROOT

- Asynchronous calls encapsulate work to be done into a Task
  - Task gets passed to TBB which runs the Task when a thread becomes available
  - When a Task finishes, it often makes another asynchronous call

- Specify the number of concurrent Events to use
  - Each Event has its own *Lane*
  - Lanes run concurrently

Lane

- Handles processing of one Event at a time



- Processing order
  - Asynchronously requests new Event from framework
    - Request goes to **Source**
  - Makes an asynchronous request to ***get*** each Data Product
    - Request goes to **Source**
  - When ***get*** completes, start asynchronous ***wait*** request to the **Waiter** assigned to the Data Product
  - When **Waiter** finishes, asynchronously signal to **Outputer** that a Data Product is ***ready***
  - Once **Outputer** is done with all data product, asynchronously signal **Outputer** the Event is ***finish***ed

Output Performance Tests

Goal

- Use realistic CMS data file
  - CMS standard ‘big’ analysis format: AOD
  - ~500 kB/event
  - Created by standard Monte Carlo simulation procedure

- Measure thread scaling of writing data from that file
  - Write ROOT format
  - Using ROOT serialization concurrently without output
  - Write a simple format

Measurements

- Machine Used
  - AMD Opteron(tm) Processor 6128
  - 4 CPUs with 8 Cores per CPU => 32 Cores in node

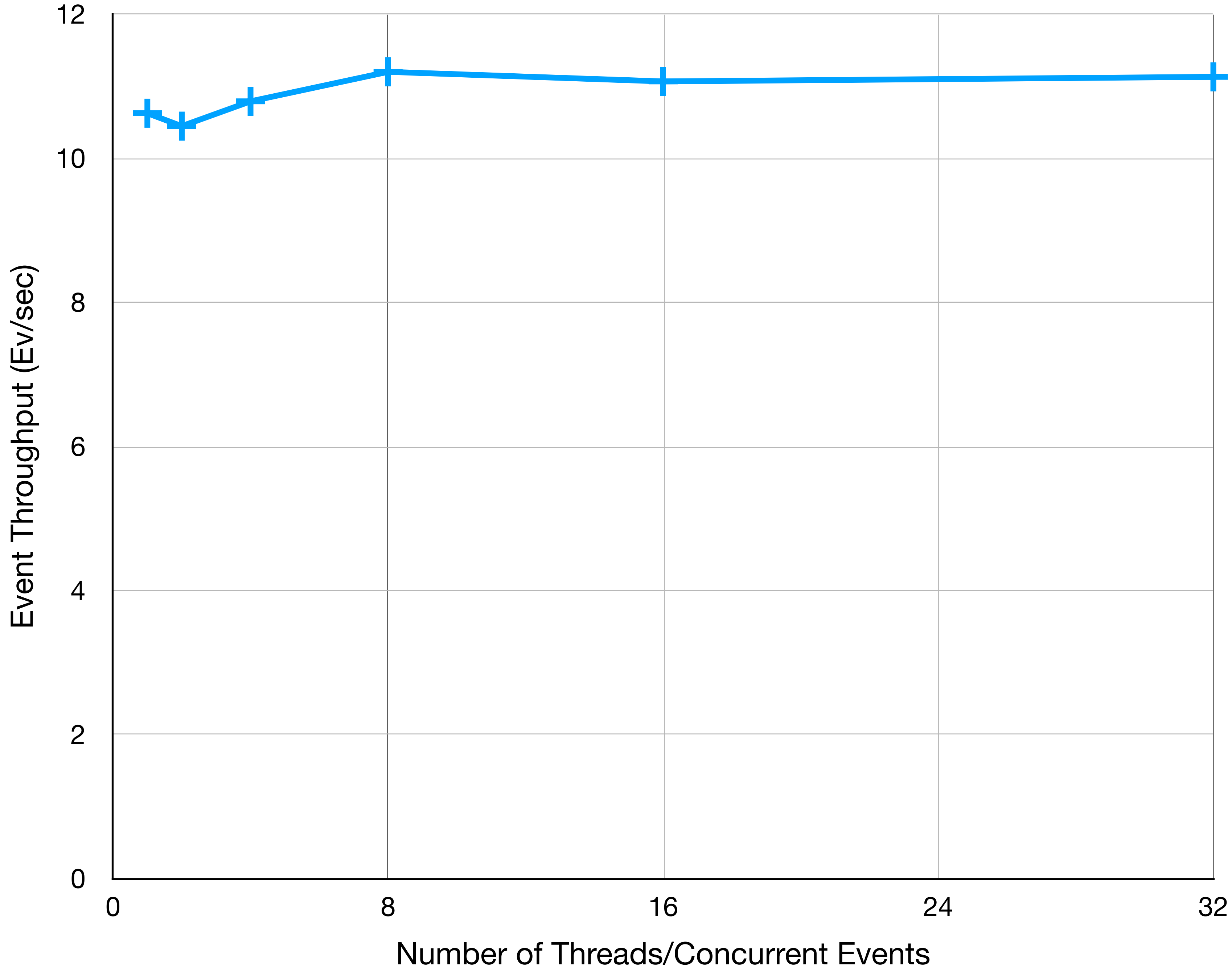
- Testing procedure
  - Number of Events processed in a job is directly proportionally to number threads used
    - Exception is when jobs stop scaling with threads, then fix number events processed
  - Unless otherwise noted, number of concurrent Events == number of threads
  - Machine was always fully loaded
    - #threads per job * # concurrently running jobs == 32

- Read first 10 events from the file and replay objects over and over
  - No dependency on storage device read speeds on measurements

- No file actually written
  - Output goes to /dev/null
  - Avoids dependency on speed of storage device in measurement

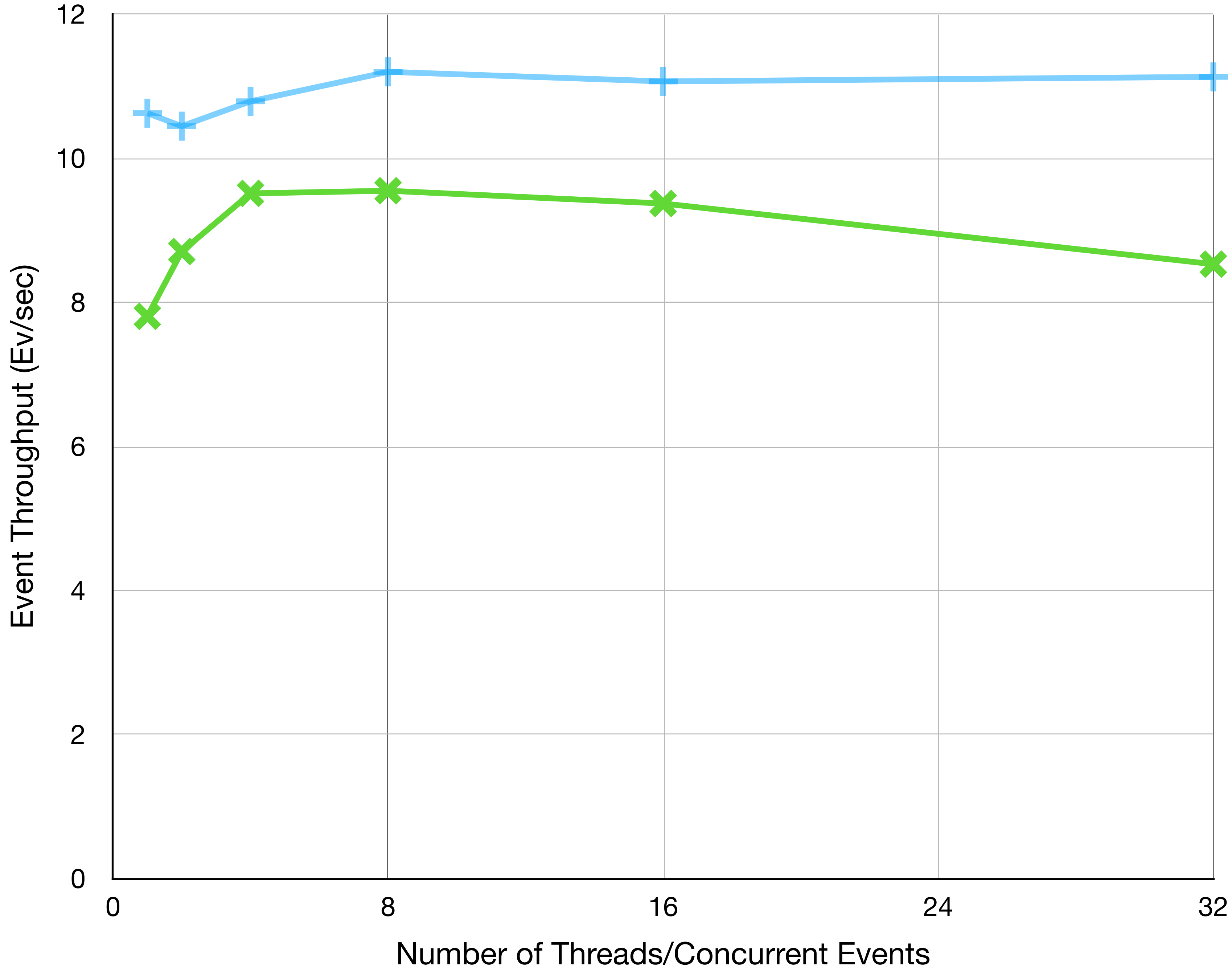
Write ROOT File No Compression

- No scaling
  - This was expected as no IMT used



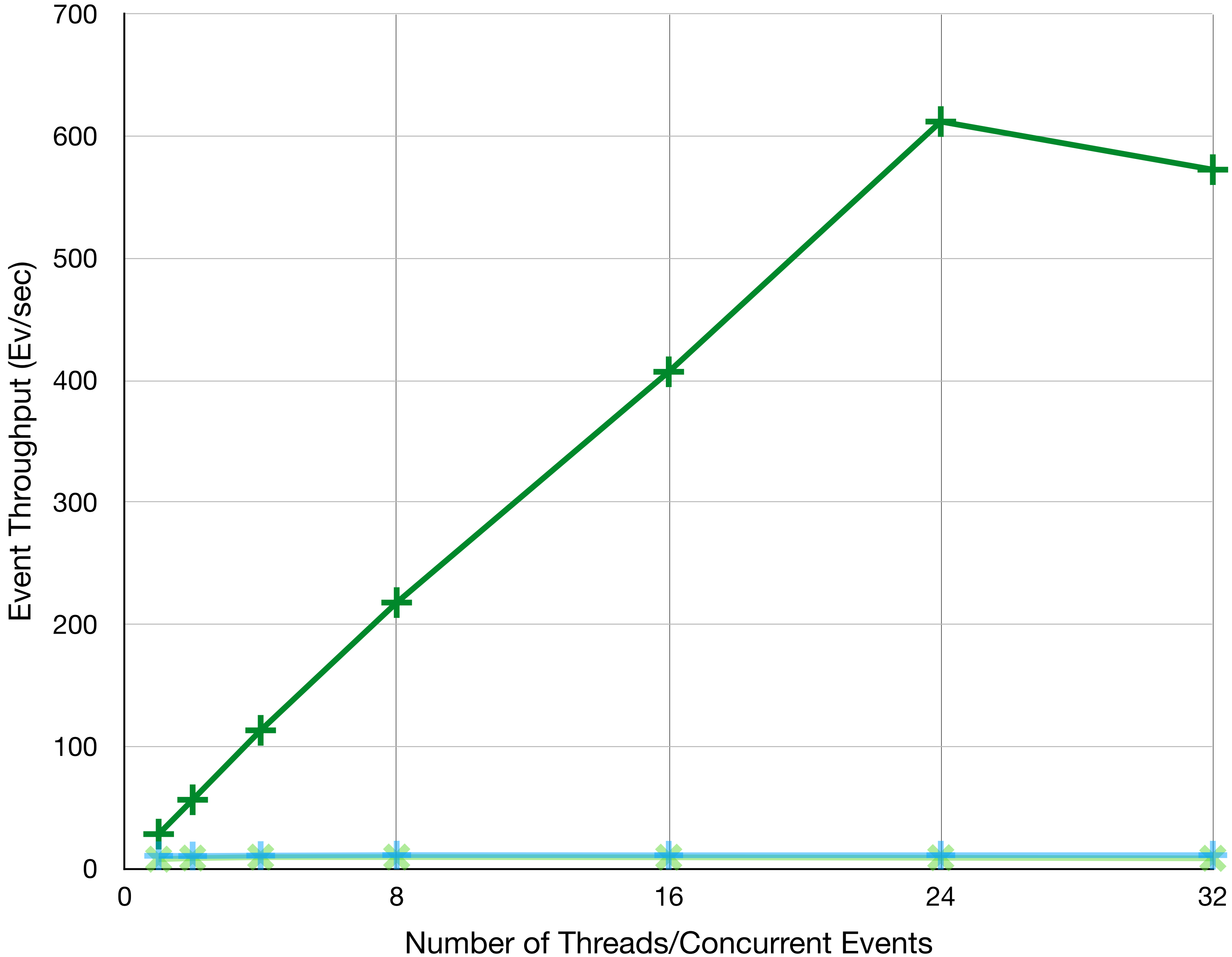
Write ROOT File With Compression

- Limited scaling
  - Not much concurrency in compressing data products



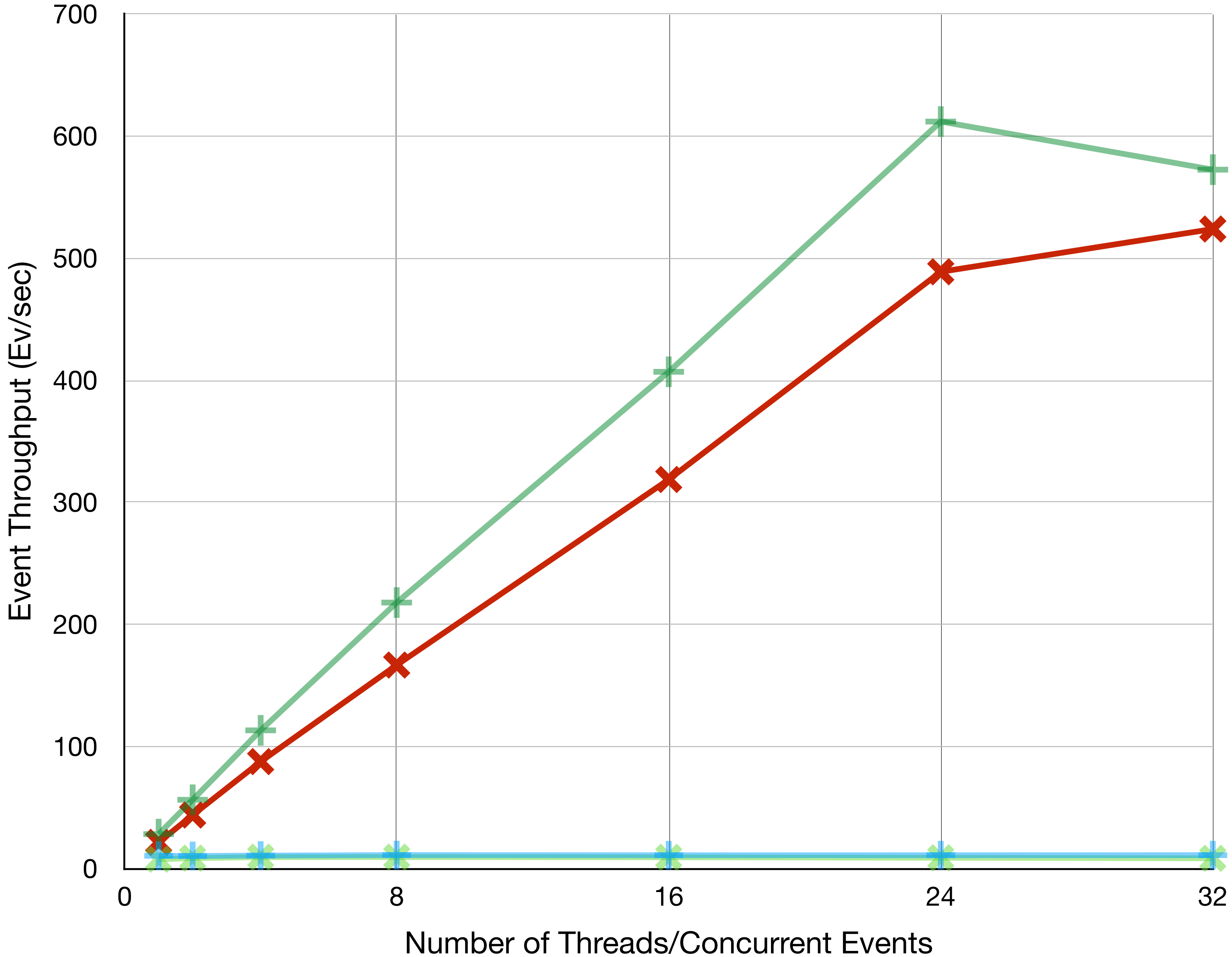
ROOT Object Serialization Only

- Good scaling till 24 threads
  - Scaling limit caused by a synchronization while serializing C++ objects



Simple Data Format

- Use ROOT C++ object serialization concurrently
  - Allow each serialized Event to be compressed concurrently



Conclusion

- Have a flexible I/O testing framework
  - Can test input and output formats and approximate HEP job timings
- Has lead to thread scaling performance of ROOT serialization
  - On second round of improvements
- Code can be found here
  - https://github.com/Dr15Jones/root_serialization